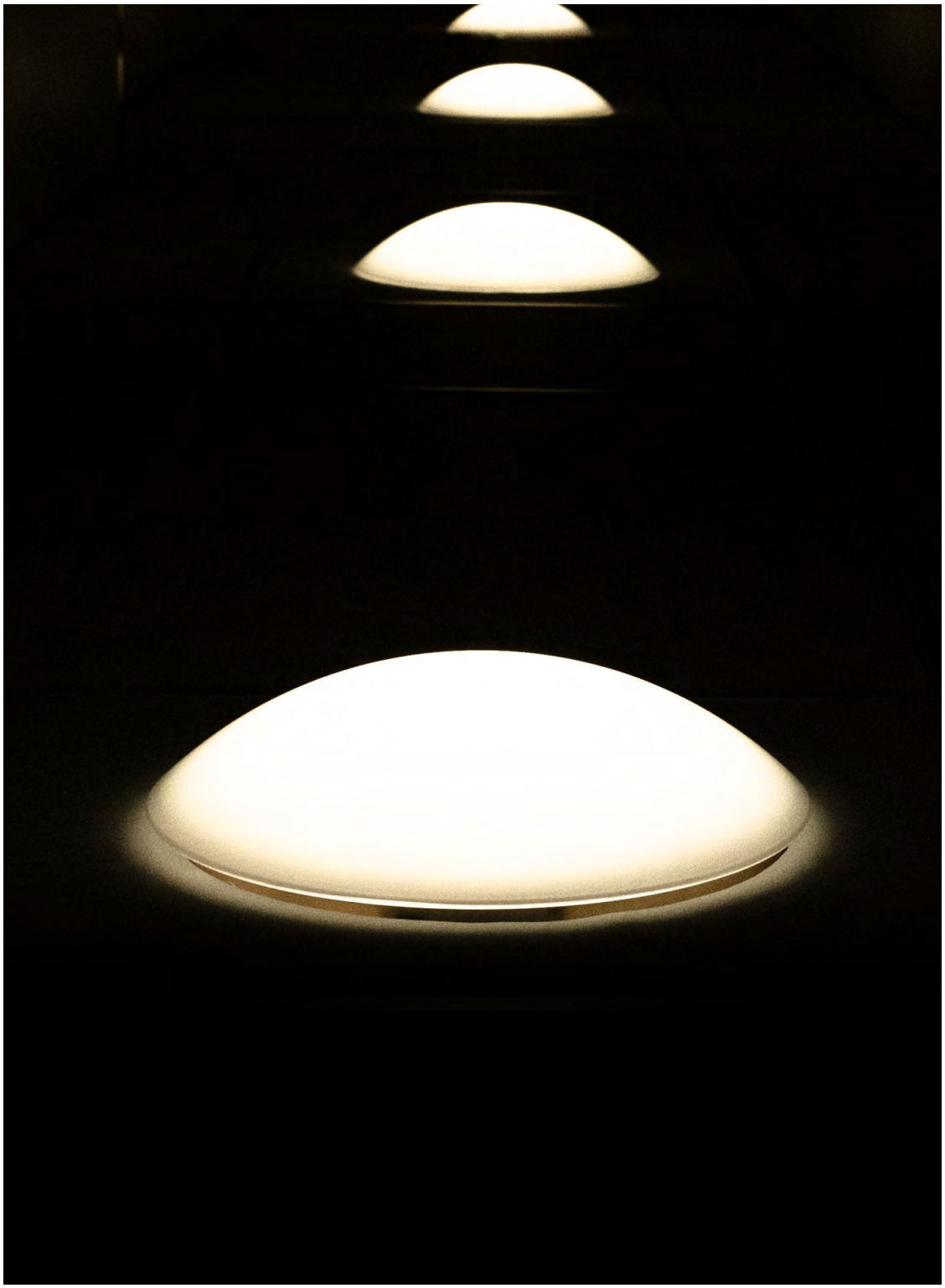

#8

# Constant Software Updates vs Public Software Acquisition

*Tommi Mikkonen*



## Abstract

Software industry has widely adopted agile software development model, where it is accepted that change is constant. Indeed, as the environment in which the software is run changes – be it changes in jurisdiction, language, user expectations, reinterpretation or requirements, or something else – the software needs to be modified to satisfy end-user needs. At the same time, many organizations, especially those that operate in the public sector, rely on tendering and detailed requirements documentation when acquiring software, with an assumption that once deployed, the software would continue to serve the end users unaltered or with minor changes covered by a maintenance contract. In this article, we will consider this fundamental mismatch from the viewpoint of what we know about software evolution in general, and then propose ways forward to design and implement public sector software that can be adapted to new, emergent needs.

# 1. Introduction

Who wouldn't be annoyed that just when there's an important task at hand, a computer – be it a smartphone, tablet, PC or cloud service – announces that unfortunately right now a software update is mandatory? After all, in just about any other business, the expectation is that at the time of transaction, complete goods are delivered, and there is no need to continuously patch it after the transaction. What makes software fundamentally different in this respect?

Software can be updated with relative ease and with low cost, at least in comparison to related hardware components. Therefore, it has become customary to adapt the software to better serve users' needs. Granted, for an individual user the changes may seem irrelevant, if the changes are made to parts that are not in active use or changes are made to improve the non-functional characteristics of the software, such as security for instance.

The fact that we know that software will constantly change is in a sharp contradiction with the practices we have for acquiring software, especially when the acquirer is a public sector organization. The acquisition is typically based on a tendering process where software providers compete to provide software that meets the associated specification. Usually, there is little or no room to negotiate over other issues than the functions of the software and the related cost. Considering changes that inevitably will be needed is easily overlooked. In fact, even phasing or incremental delivery of the system is often not considered, as many organizations find such deployment unthinkable and risky (Koski, 2019).

This article addresses the apparent contradiction between continuous software development and public software, where large systems are typically acquired – for instance, Apotti [1], an information system for merging social care and health care services and to standardize associated operational routines for Helsinki Metropolitan region, Finland, has been estimated to cost over 800 Me (Wikipedia, 2023). The system was acquired by a plan-first approach, but it has a phased SW delivery, with new features and expanding user base in each phase.

First, the article will provide background on software evolution and techniques that companies apply to support continuous change. Next, we provide an overview to today's practices for public tendering. Finally, proposals are made to transform public sector actors to software organizations, so that they can truly benefit from the digital transformation.

# 2. Basics of Software Evolution

The origins of the so-called Lehman's laws of software evolution are in the work of Meir 'Manny' Lehman, who categorized software to three flavors, called S, P and E type software (Lehman, 1996). Of these, S type software is based on a precise specification, such as a mathematical formula. P type software does not have such an exact specification, but it always performs certain well-defined operations that determine everything the software can ever

1 https://www.apotti.fi/en/

do – such as chess game moves. E type software, in contrast, are part of real-world processes and are inextricably linked to their environment.

Based on the above, while S and P type software can be designed to satisfy the whole specification or design space, E type software requires constant fine-tuning to serve users and operate in a changing context, constituted by the connection to the real world. This continuous fine-tuning leads to continuously increasing complexity, which in the case of software typically means decomposing the software to smaller pieces, each of which can be worked on separately.

"Considering changes that inevitably will be needed is easily overlooked."

## 3. Software Solutions for Supporting Software Evolution

Due to the omnipresent nature of change in software, a large part of software engineering has been the invention of technical and methodological solutions for managing and organizing increasing complexity. Examples are many. Modularity allows changing internals of a module without harming other modules that rely on its services (Parnas, 1972). Inheritance enables making new variants of a baseline module that differ from the baseline a bit or combine things that go hand in hand in design (Taivalsaari, 1996). Virtual machines allow running the same software on different hardware (Goldberg, 1974). Containers enable bundling together pieces of software that implement a coherent whole that can be deployed independently (Koskinen et al., 2019). Microservice architecture (Nadareishvili et al., 2016) provides instructions how such containers should be best used, to flexibly introduce new configurations, and so on.

With the new infrastructure, releasing new software has become easier and cheaper. Today, instead of releases that were made at regular intervals, say every six months, which were common in the past, new software can be deployed at the very moment it is considered ready for operational use – at very least for a selected population of users. This has been made possible by new development approaches, such as continuous software engineering (Fitzgerald and Stol, 2017) and DevOps (Ebert et al., 2016). Both require maintaining a sophisticated deployment pipeline (Humble and Farley, 2010), but in contrast to investing the effort to each time software is released, this maintenance task is considered smaller. While in general, rapid speed of software development has been considered as a negative thing for software quality, it has been shown that this need not be the case, as many of the quality assurance actions can be automated as well (Tamburri & Perez-Palacin, 2018). Moreover, if a new release is faulty, it is possible to return to using an older version that has been validated in use using the same release mechanisms.

## 4. Public Sector Software: S, P, or E type?

Today, there are large organizations that acquire large software systems, with an expectation that the acquired system will serve the organization unchanged for a long time, if software requirements are sufficiently well and precisely formulated. In fact, creating a requirements specification is often an effort of its own. Then, when this task is completed, the requirements are used as basis for tendering, or placing an open offer for software vendors to fulfill.

Tendering is the process where an organization that needs a software system invites bids for software projects, to be delivered within a fixed deadline (Koski 2019). The process starts by describing a problem the acquiring organization has and outlining a project with which to solve the problem. The acquiring organization concentrates on describing the problem they have and ask potential suppliers to describe ways to meet the requirements, together with an estimate on pricing. Then, the suppliers study the requirements and propose a solution they see fit for satisfying the requirements.

Based on the proposals, one of the suppliers is selected, based on price, assumed quality of the proposal, or a weighted combination of the two. Sometimes, politics, personal contacts, track record, and other non-technical aspects also have an impact on the selection, although tendering should be independent from such and only focus on the offering made by the potential suppliers. After the closure of the tendering process itself, the actual development will begin. More time is spent on the further specification, design, development, testing, releasing, and eventually resulting in the deployment of the system to its operative environment.

Because of the time and cost of the tendering process, it often happens that organizations do not want to change the software they have, but rather change the behavior of the people. In terms of software evolution, this means that while everyone understands that the software in question is of type E – living and evolving with its environment – the organization that uses it does its best to make it an S or P type of software, by changing the way the organization works, thus omitting the changes to the software. Moreover, while tendering the acquiring organization often specifies its current ways of working, instead of critically assessing those to improve the underlying processes.

Typically, the system being purchased is a prerequisite for the buyer's core functions, such as patient management in the case of a hospital, for instance. Purchasing such a core system from a subcontractor introduces a fundamental trust relationship to the software provider, who therefore must be a well-established company to manage risks related to the relationship. This in essence eliminates all small companies from the business. Furthermore, because the system is question is tailor-made, it is impossible to consider similar systems from another supplier, so there is little room for new partnerships. Therefore, there are two paths to consider – either pay extra for every new feature in the software that runs everyday operations to the selected vendor,

who is the only game in town and takes that into account in pricing, or, alternatively, adapt everyday functions to the software as it is when it was purchased. Both are costly operations.

## 5. Macro-services to the rescue

Because it is next to impossible to acquire large systems as E-type systems, one should consider scaling down the expectations. Instead of acquiring one, massive system that is completely managed by the vendor that delivering it, it is possible to decompose the needs to a collection of subsystems, each consisting of functions of meaningful size. These subsystems can be regarded as macroservices (Setälä et al., 2021), in analogy to microservices mentioned above. However, while microservices are the smallest meaningful operational units in the implementation sense, macroservices are the smallest meaningful software systems to be tendered and subcontracted. Then, each of the services can evolve separately, and the complete system, constituted by all the necessary macroservics, will remain in operation, even if some of the macroservices are updated or even replaced by new versions, provided by different vendors.

"Because it is next to impossible to acquire large systems as E-type systems, one should consider scaling down the expectations."

While the majority of the macroservices would be built to order, some of them might be services from other organizations. Often, this is the case already today, as for example from the Digital and Population Information Agency provides services to several public organizations, and their services need to be compatible with each other. Each macroservice can then evolve, maintaining their E type software status, on their own terms, if the interface to the services they provide remains unaltered or backward compatible. Sometimes, this is what takes place behind the curtains when acquiring a monolithic application built to order, but the vendor who provides the system will not reveal the details to the client or consider the implications in pricing. An example of such is sketched in (Ghezzi et al. 2023).

The trend towards macroservices has also been noticed at the EU scale. The Gaia-X initiative (Braud et al., 2021) is an approach that aims at ensuring data sovereignty and defining of data usage constraints, which effectively means leaving single-vendor, monolithic software systems, and entering an era where systems are built out of services that can be adapted to different use cases. However, as Gaia-X is still at an initial phase, its role as a cradle of microservice development remains unclear.

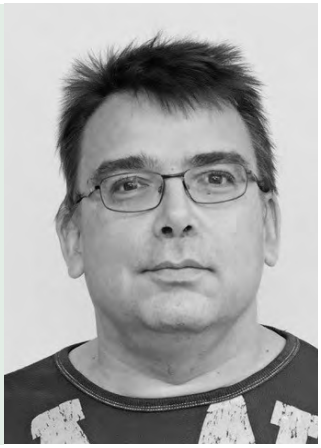
## 6. Conclusions

While software in operational use is constantly updated and modified to better serve users' needs, organizations that are bound by legislation to invite tenders must seek ways to enable software evolution. This evolution can take place in two ways, either behind the curtains by the vendor that delivers the software and charges for every change, or transparently by selecting an architecture where individual parts of the system can be replaced. The former leaves the acquiring organization at the mercy of the vendor, whereas the latter provides more freedom to operate – if the acquiring organization can take responsibility for managing the evolution of the system. To improve awareness of this choice, one alternative is to change the procurement law that bounds public organizations, so that they demonstrate awareness of the evolving nature of software and are prepared to face it in systems they acquire. Otherwise, the history will simply repeat itself, and we end up paying more and more for our public services.

Kirjoittaja

**Tommi Mikkonen**

Tommi Mikkonen on ohjelmistotekniikan professori Jyväskylän yliopiston Informaatioteknologian tiedekunnassa. Hänen tutkimusintressinsä keskittyvät ohjelmistotuotannon menetelmiin ja käytännön ohjelmistotyöhön.



# References


Braud, A., Fromentoux, G., Radier, B., & Le Grand, O. (2021). The Road to European Digital Sovereignty with Gaia-X and IDSA. *IEEE Network*, 35(2), 4-5.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). *DevOps. IEEE Software*, 33(3), 94-100.

Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176-189.

Ghezzi, R., Koski, A., Lautanala, J., Lehtisalo, M., Mikkonen, T. and Setälä, M. Towards Sustainable Software for Public Sector Information Systems. In *Proceedings of ICSSP'23*, IEEE, 2023.

Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, 7(6), 34-45.

Humble and D. Farley. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Pearson Education.

Koski, A. (2019). *On the Provisioning of Mission Critical Information Systems based on Public Tenders*. Doctoral Dissertation, University of Helsinki, Finland.

Koskinen, M., Mikkonen, T., & Abrahamsson, P. (2019). Containers in Software Development: A Systematic Mapping Study. In I*nternational Conference on Product-Focused Software Process Improvement* (pp. 176-191). Springer, Cham.

Lehman, M. M. (1996). Laws of software evolution revisited. In *European Workshop on Software Process Technology* (pp. 108-124). Springer, Berlin, Heidelberg.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering* (pp. 479-498). Springer, Berlin, Heidelberg.

Setälä, M., Abrahamsson, P., and Mikkonen, T. (2021). Elements of Sustainability for Public Sector Software–Mosaic Enterprise Architecture, Macroservices, and Low-Code. In *International Conference on Software Business* (pp. 3-9). Springer, Cham.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys* (CSUR), 28(3), 438-479.

Tamburri, D. A., & Perez-Palacin, D. (2018). DevOps Quality Engineering. *Journal of Software: Evolution and Process*, 1-3.

Wikipedia. (2023) Apotti (potilastietojärjestelmä). Available at https://fi.wikipedia.org/wiki/Apotti_(potilastietojärjestelmä) (referenced Sept. 18, 2023)